Writing the code is the easy part of solving this problem. Once we went from the vague statement of a problem about bunnies to a set of recursive equations, the code almost wrote itself. Finding some kind of abstract way to express a solution to the problem at hand is very often the hardest step in building a useful program. We will talk much more about this later in the book.

As you might guess, this is not a perfect model for the growth of rabbit populations in the wild. In 1859, Thomas Austin, an Australian farmer, imported twenty-four rabbits from England, to be used as targets in hunts. Ten years later, approximately two million rabbits were shot or trapped each year in Australia, with no noticeable impact on the population. That's a lot of rabbits, but not anywhere close to the 120th Fibonacci number.[31]

Though the Fibonacci sequence does not actually provide a perfect model of the growth of rabbit populations, it does have many interesting mathematical properties. Fibonacci numbers are also quite common in nature.

Finger exercise: When the implementation of `fib` in Figure 4.7 is used to compute `fib(5)`, how many times does it compute the value of `fib(2)` on the way to computing `fib(5)`?

4.3.2 Palindromes

Recursion is also useful for many problems that do not involve numbers. Figure 4.8 contains a function, `isPalindrome`, that checks whether a string reads the same way backwards and forwards.

The function `isPalindrome` contains two internal **helper functions**. This should be of no interest to clients of the function, who should care only that `isPalindrome` meets its specification. But you should care, because there are things to learn by examining the implementation.

The helper function `toChars` converts all letters to lowercase and removes all non-letters. It starts by using a built-in method on strings to generate a string that is identical to `s`, except that all uppercase letters have been converted to lowercase. We will talk a lot more about **method invocation** when we get to classes. For now, think of it as a peculiar syntax for a function call. Instead of putting the first (and in this case only) argument inside parentheses following the function name, we use **dot notation** to place that argument before the function name.

[31] The damage done by the descendants of those twenty-four cute bunnies has been estimated to be $600 million per year, and they are in the process of eating many native plants into extinction.

```
def isPalindrome(s):
   """Assumes s is a str
      Returns True if letters in s form a palindrome; False
        otherwise. Non-letters and capitalization are ignored."""

   def toChars(s):
      s = s.lower()
      letters = ''
      for c in s:
        if c in 'abcdefghijklmnopqrstuvwxyz':
            letters = letters + c
      return letters

   def isPal(s):
      if len(s) <= 1:
        return True
      else:
        return s[0] == s[-1] and isPal(s[1:-1])

   return isPal(toChars(s))
```

Figure 4.8 Palindrome testing

The helper function isPal uses recursion to do the real work. The two base cases are strings of length zero or one. This means that the recursive part of the implementation is reached only on strings of length two or more. The conjunction[32] in the else clause is evaluated from left to right. The code first checks whether the first and last characters are the same, and if they are goes on to check whether the string minus those two characters is a palindrome. That the second conjunct is not evaluated unless the first conjunct evaluates to True is semantically irrelevant in this example. However, later in the book we will see examples where this kind of **short-circuit evaluation** of Boolean expressions is semantically relevant.

This implementation of isPalindrome is an example of an important problem-solving principle known as **divide-and-conquer**. (This principle is related to but slightly different from divide-and-conquer algorithms, which are discussed in Chapter 10.) The problem-solving principle is to conquer a hard problem by breaking it into a set of subproblems with the properties that

- the subproblems are easier to solve than the original problem, and
- solutions of the subproblems can be combined to solve the original problem.

[32] When two Boolean-valued expressions are connected by "and," each expression is called a **conjunct**. If they are connected by "or," they are called **disjuncts**.

Divide-and-conquer is a very old idea. Julius Caesar practiced what the Romans referred to as *divide et impera* (divide and rule). The British practiced it brilliantly to control the Indian subcontinent. Benjamin Franklin was well aware of the British expertise in using this technique, prompting him to say at the signing of the U.S. Declaration of Independence, “We must all hang together, or assuredly we shall all hang separately.”

In this case, we solve the problem by breaking the original problem into a simpler version of the same problem (checking whether a shorter string is a palindrome) and a simple thing we know how to do (comparing single characters), and then combine the solutions with `and`. Figure 4.9 contains some code that can be used to visualize how this works.

```
def isPalindrome(s):
   """Assumes s is a str
      Returns True if s is a palindrome; False otherwise.
       Punctuation marks, blanks, and capitalization are ignored."""

   def toChars(s):
      s = s.lower()
      letters = ''
      for c in s:
        if c in 'abcdefghijklmnopqrstuvwxyz':
            letters = letters + c
      return letters

   def isPal(s):
      print('  isPal called with', s)
      if len(s) <= 1:
         print('  About to return True from base case')
         return True
      else:
         answer = s[0] == s[-1] and isPal(s[1:-1])
         print('  About to return', answer, 'for', s)
         return answer

   return isPal(toChars(s))

def testIsPalindrome():
   print('Try dogGod')
   print(isPalindrome('dogGod'))
   print('Try doGood')
   print(isPalindrome('doGood'))
```

Figure 4.9 Code to visualize palindrome testing

When testIsPalindrome is run, it will print

```
Try dogGod
  isPal called with doggod
  isPal called with oggo
  isPal called with gg
  isPal called with
  About to return True from base case
  About to return True for gg
  About to return True for oggo
  About to return True for doggod
True
Try doGood
  isPal called with dogood
  isPal called with ogoo
  isPal called with go
  About to return False for go
  About to return False for ogoo
  About to return False for dogood
False
```

4.4 Global Variables

If you tried calling `fib` with a large number, you probably noticed that it took a very long time to run. Suppose we want to know how many recursive calls are made? We could do a careful analysis of the code and figure it out, and in Chapter 9 we will talk about how to do that. Another approach is to add some code that counts the number of calls. One way to do that uses **global variables**.

Until now, all of the functions we have written communicate with their environment solely through their parameters and return values. For the most part, this is exactly as it should be. It typically leads to programs that are relatively easy to read, test, and debug. Every once in a while, however, global variables come in handy. Consider the code in Figure 4.10.

In each function, the line of code `global numFibCalls` tells Python that the name `numCalls` should be defined at the outermost scope of the module (see Section 4.5) in which the line of code appears rather than within the scope of the function in which the line of code appears. Had we not included the code `global numFibCalls`, the name `numFibCalls` would have been local to each of the functions `fib` and `testFib`, because `numFibCalls` occurs on the left-hand side of an assignment statement in both `fib` and `testFib`. The functions `fib` and `testFib` both have unfettered access to the object referenced by the variable `numFibCalls`. The func-